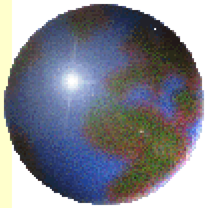
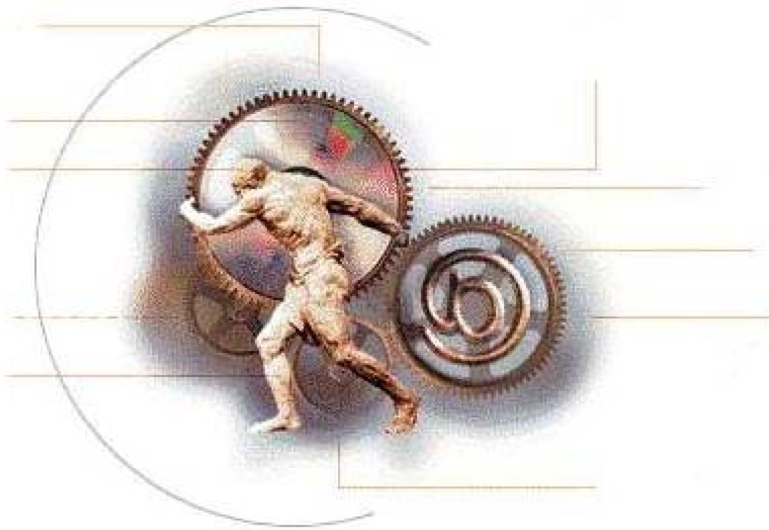
資料結構(Data Structures)

Course 8: Advance Tree (高等樹)





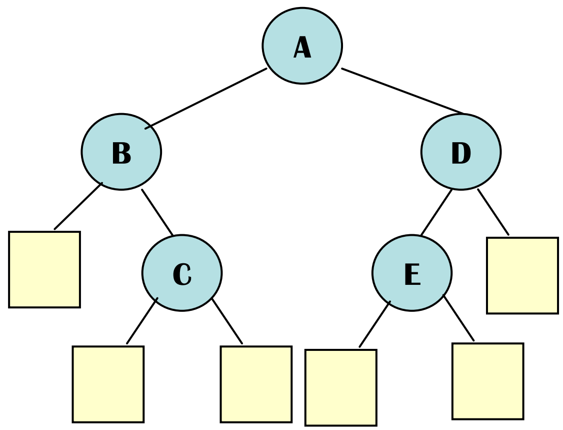
授課教師:陳士杰

國立聯合大學 資訊管理學系

- 本章重點
  - Extended Binary Tree
    - Min-weighted External Path Length
  - Binary Search Tree的效益
  - AVL Tree
  - M-way Search Tree
  - B Tree
  - B^+ Tree

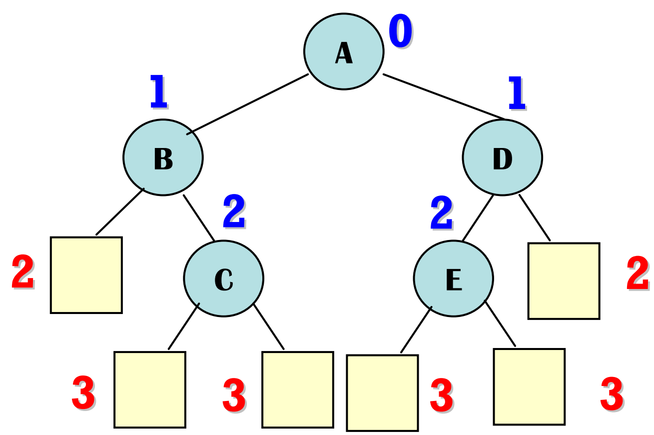
Extended Binary Tree (延伸二元樹)

- Def: 具有**External Node**的B.T.稱之。
  - 一個二元樹具有n個節點，若以Link List表示，則會有 n+1 條空Link，在這些空Link上加上特定節點，稱為External Node (或Failure Node)，其餘Nodes稱Internal Nodes。
  - 範例：



  - 外部節點數 = 內部節點數 + 1　　(P.S.: 2n – (n-1)=n+1)
  - 為何外部節點又被稱為失敗節點？
    - 以二元搜尋樹的角度來說，若搜尋到外部節點時，代表在原本的二元樹中找不到想要的資料，也就是搜尋失敗。

- Def:
  - $I = \sum_{i=1}^{n}$(Root 到內部節點 i 的路徑長度)
  - $E = \sum_{j=1}^{n+1}$(Root 到外部節點 j 的路徑長度)
- 定理 : E = I + 2n (n為內部節點個數)
- 範例 1 : 一個具有5個Internal Nodes的 Extended B.T., 其 I值與E值分別為?

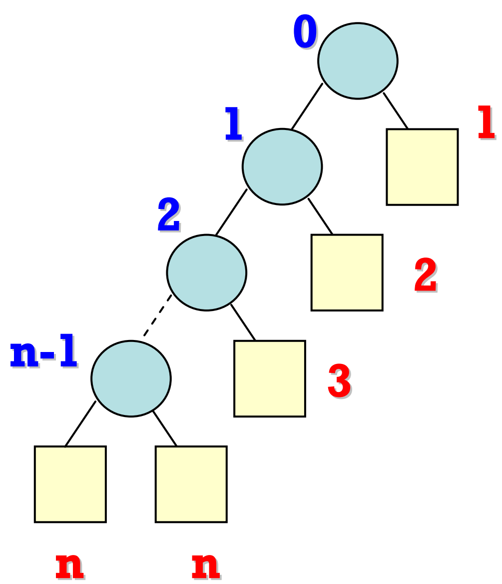


I = 0+1+1+2+2 = 6

E = 2+2+3+3+3+3= 16

=I +2n

- 範例 2：一個具有n個Internal Nodes的Skewed Extended B.T.，其I值與E值分別為?



$I = 0+1+2+\ldots+(n-1) = [(n-1)n]/2$

$E = 1+2+3+\ldots+2n = [(n-1)n]/2 + 2n$

$= [n(n+3)]/2$

$= I + 2n$

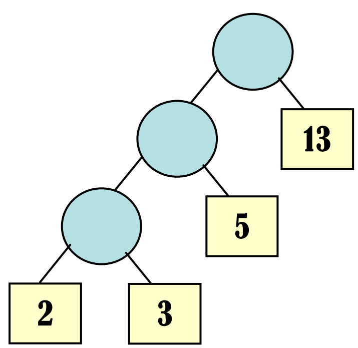
- 結論：
  - E值與I值成正比
  - 愈平衡 (即：高度愈小) 的Extended B.T.，其E值與I值愈小
  - 然而，若外部Node有加權值時，則第二個結論不見得成立!!

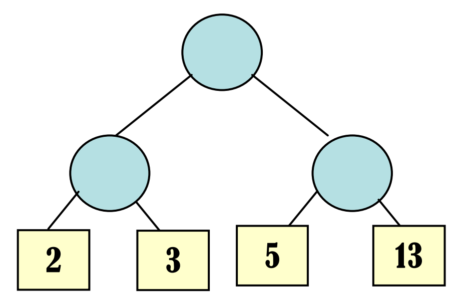
Weighted External Path Length (W.E.P.L.; 加權外部路徑長度)

- Def: Extended B.T.若有n個內部節點，則會有 n+1 個外部節點。分別給予每一個外部節點 1 個加權值，則：

$$\text{W.E.P.L.} = \sum_{j=1}^{n+1} \left[\begin{array}{l} (\text{Root 到外部節點 } j \text{ 的路徑長度}) \times \\ (\text{外部節點 } j \text{ 的加權值}) \end{array} \right]$$

- 範例：





W.E.P.L. = 3 × 2+3 × 3+2 × 5+1 × 13

=38

W.E.P.L. = 2 × (2+3+5+13)

=46

- 當節點有加權值，且每個加權值不盡相同時，則當樹愈平衡時，不見得其外部路徑長度就愈小!!
- 問題：
  - 若有三個 (n個) 內部節點，什麼情況它們的Weighted E.P.L.是最小的?

Min. W.E.P.L. (最小加權外部路徑長度)

- Def:
  - 給予(n+1)個外部節點加權值，在 $C_n^{2n}/(n+1)$顆樹中，具有最小的W.E.P.L.稱之。
- 主要應用:
  - (n+1)個訊息傳輸，其平均解碼時間最小 (或：編碼位元長度最小)

- 求Min. W.E.P.L.的方法有兩種:
  - Brute Force法 (暴力法)
  - Huffman Algorithm (霍夫曼演算法)

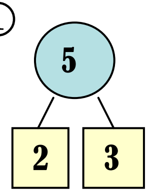
- 令W為外部節點加權值的集合，Huffman Algo.的執行步驟如下：
  - 自W中取出2個具最小加權值的節點
  - 替這2個節點建立Extended B.T.
  - 再將此2個節點的加權值和加入W中
  - Repeat前三步直到W中只剩一個加權值為止

- 範例：有6個外部節點，其加權值分別為：2, 3, 5, 7, 9, 13。求Min. W.E.P.L.。

Ans: W = {2, 3, 5, 7, 9, 13}

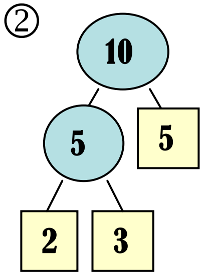
(數值放左leaf或放右leaf對於找min. W.E.P.L.無影響，
但習慣上將小的放在左邊)

①



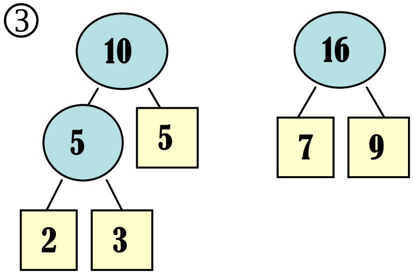
⇨ W = {5, 5, 7, 9, 13}

②



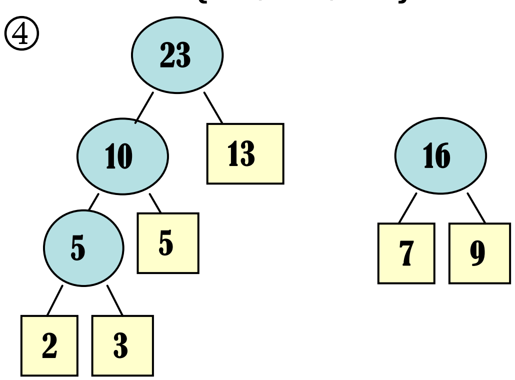
⇨ W = {10, 7, 9, 13}

③



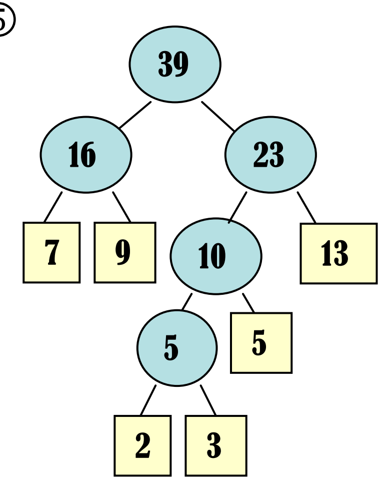
⇨ W = {10, 16, 13}

④



⇨ W = {23, 16}

⑤



⇨ W = {39}

- **Huffman Tree**
- Min. W.E.P.L. = 93

(2×7+2×9+2×13+3×5+4×2+4×3)

- Huffman Algo.的精神：
  - 若希望求得加權路徑長度總和為最小，則 “加權值愈重的節點，離Root要愈近” !!
  - ∴建樹時是從底層建起；在建樹的過程中，皆是取當時在W集合中<u>權重值最輕</u>的兩個節點。

- 【應用 1】有6個Message要傳輸，其出現頻率如下：

$$M_1 = \frac{2}{39}，M_2 = \frac{3}{39}，M_3 = \frac{5}{39}，M_4 = \frac{7}{39}，M_5 = \frac{9}{39}，M_6 = \frac{13}{39}$$

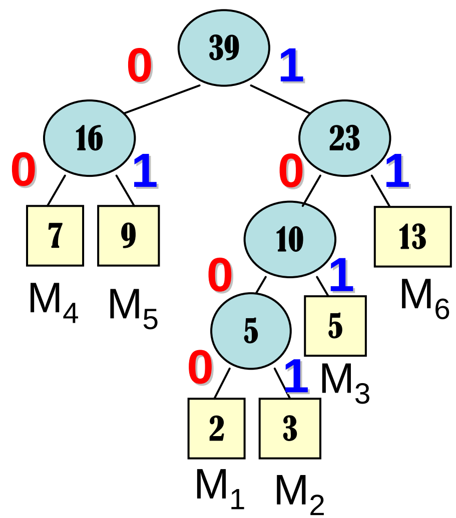
今希望平均解碼時間最小，則：

  - 解碼時間最短的Encoding/Decoding Tree為何?
  - 各Message之編碼內容為何?
  - 平均最小編碼位元長度為何?

Ans：

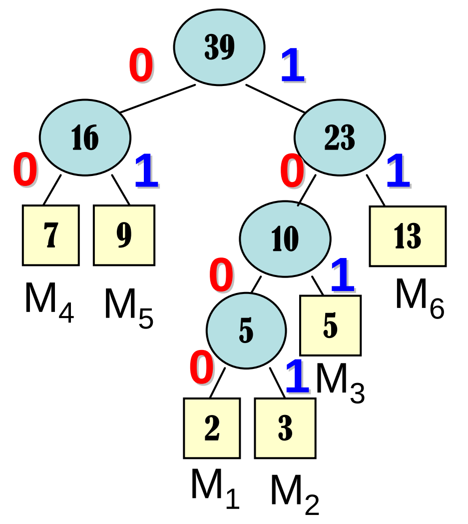
  - Message：外部節點；出現頻率：加權值

- 解碼時間最短的Encoding/Decoding Tree：
  - 將出現頻率通分後，以分子當加權值建立Huffman Tree



  - Encoding/Decoding Tree (都是同一顆Tree):
    - 左分支:0
    - 右分支:1

- 各Message之編碼內容: (由Root開始走)
  - $M_1 = 1000$
  - $M_2 = 1001$
  - $M_3 = 101$
  - $M_4 = 00$
  - $M_5 = 01$
  - $M_6 = 11$



- Message Decoding Time = Message Bit數

  = Root到外部節點的路徑長度
- 經常出現的訊息, Bit數要愈少; 即路徑長度要愈短。

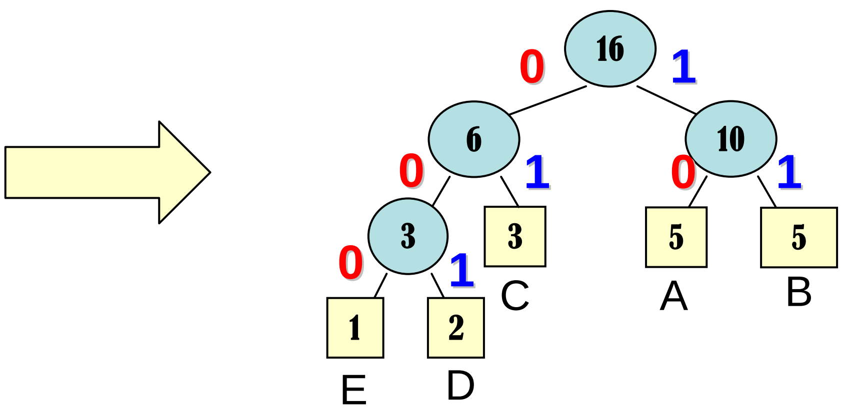
- 平均最小編碼位元長度：
  - Σ(編碼位元數×出現頻率)

$$2\times\frac{7}{39}+2\times\frac{9}{39}+2\times\frac{13}{39}+3\times\frac{5}{39}+4\times\frac{3}{39}+4\times\frac{2}{39}=\frac{93}{39}\doteqdot 2.38$$

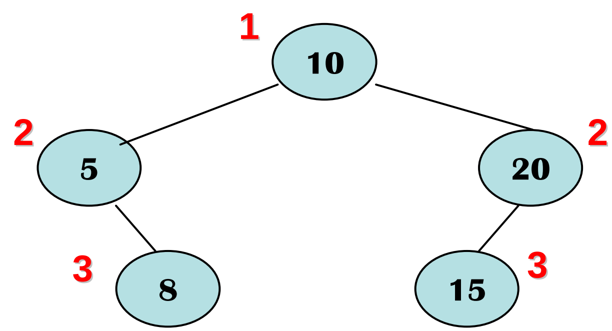
- 【應用 2】有一字串AABBBACCBADDECBA，求Encoding/ Decoding Tree為何?

Ans：

- 每個字元的出現頻率：
  - A⇨5次
  - B⇨5次
  - C⇨3次
  - D⇨2次
  - E⇨1次



- 當B.S.T.建立之後，可以用來搜尋資料。因此，由平均比較次數來決定其效益。
  - 若平均比較次數愈低，則效益愈高
- 例 1：
  - 成功搜尋的平均比較次數：
    (1+2+2+3+3)/5 = 11/5 = 2.2



- 結論：不考慮加權值或加權值皆相同的情形下
  - 愈平衡的Binary Search Tree，其平均比較次數愈小，Big-O≈ O(log n)。
  - 愈偏斜的Binary Search Tree，其平均比較次數愈大，Big-O ≈ O(n)。

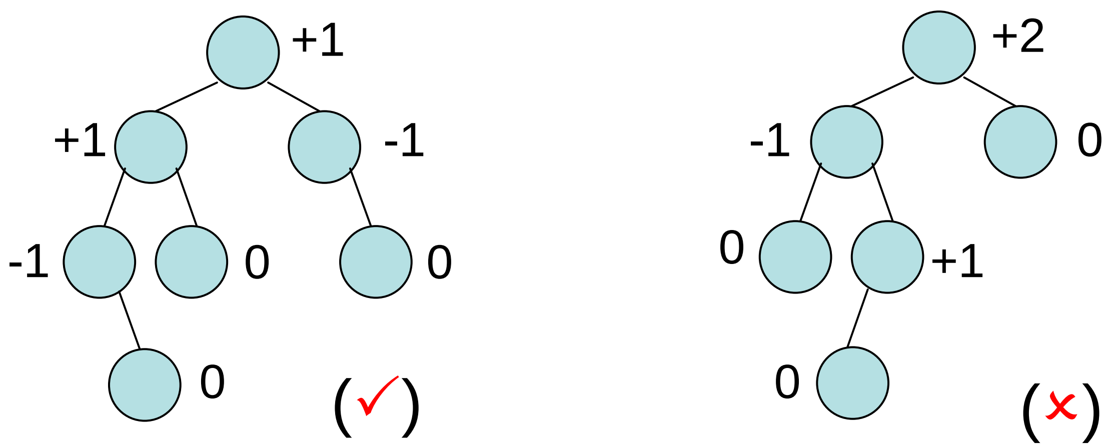
- 在靜態的環境下，一般Binary Search Tree的建構方法可以有較充裕的時間將欲搜尋的所有資料建構成較平衡的Binary Search Tree，以提升後續資料搜尋時的效率。
- 然而，若是在動態的環境下，資料可以隨時Insert / Delete。此時，一般Binary Search Tree的建構方法會產生Skewed Binary Tree，導致資料搜尋的效率變差 (即：O(n))。
- 需要AVL Tree，面對動態資料，Tree的高度均維持在高度平衡的情況。

■ AVL Tree (高度平衡樹)

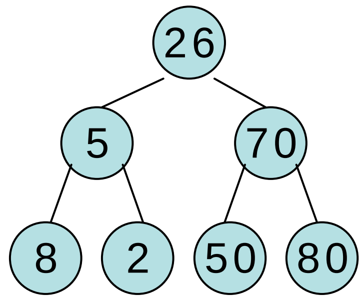
- Def: 為一個Height Balance的Binary Search Tree，可以為空，若不為空，則滿足：
  - $|h_L - h_R| \le 1$，其中 h_L 與 h_R 為左、右子樹的高度。
  - 左右子樹亦是AVL Tree (∴AVL Tree也有遞迴的特性)

（前題：不考慮加權值或加權值皆相同的情形下）

- 範例 1：下列Binary Search Tree何著為AVL Tree?



- 範例 2：下列Binary Tree是否為AVL Tree?



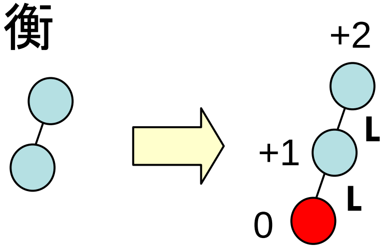
(×)

  - ∵ 8 > 5，違反了Binary Search Tree的條件，∴不是AVL Tree

- Balance Factor (平衡因子) of a node：
  - Def: B.F.= h_L - h_R
  - 在AVL Tree中，所有節點的B.F.只有三種值：-1，0，+1

AVL Tree 不平衡的情況

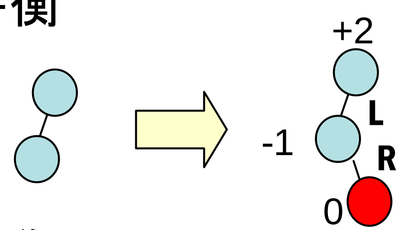
- 建構AVL Tree的過程中，當插入一個新的節點時，可能會造成以下四種不平衡的情況：
  - LL不平衡



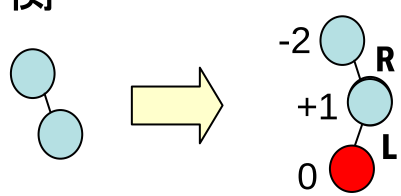
⇒ 離新進節點最近的不平衡點

因為新進節點是在此最近不平衡點的LL方向，故稱為LL不平衡 (只看兩階)

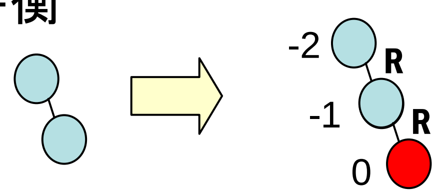
  - LR不平衡



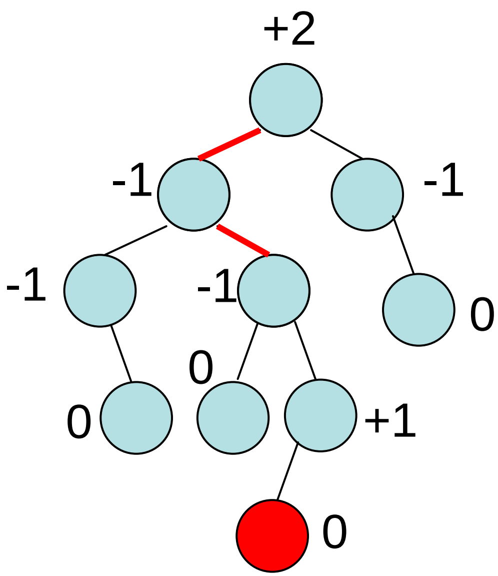
  - RL不平衡



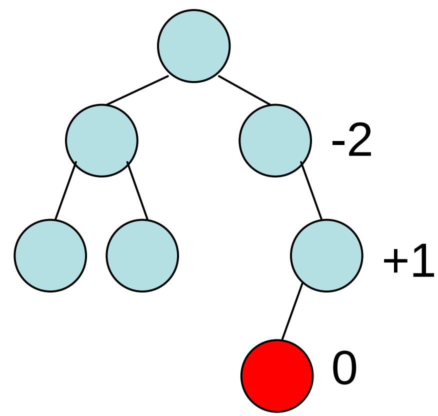
  - RR不平衡



- 以下AVL Tree插入新節點後，會產生何種不平衡?



LR不平衡
(只看兩階)

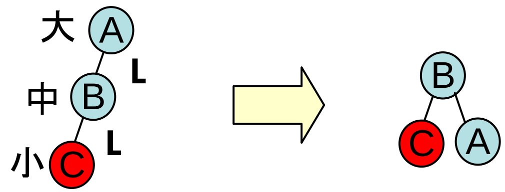


RL不平衡

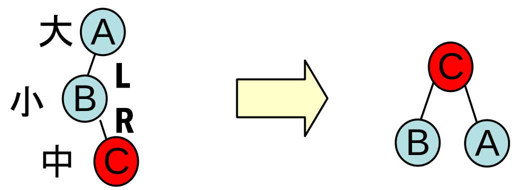
AVL Tree 不平衡情況的調整方式

- 原則：
  - 由於四種不平衡情況會牽涉到三個節點，且三個節點內存放的值會有小、中、大三種不同的資料。故調整時，三個節點的<u>中間值往上拉</u>，<u>小的值放左</u>，<u>大的值放右</u>。
  - 調整後，若產生沒有父親的子樹，則以<u>二元搜尋樹插入資料</u>的方式來判斷該放哪裡。

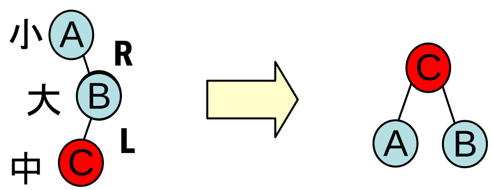
- 範例 1:若有以下四種不平衡的情況, 該如何調整:
  - LL不平衡



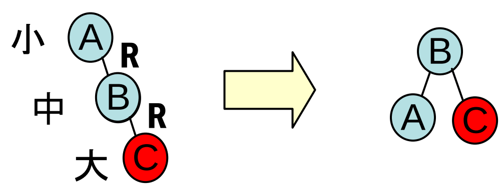
  - LR不平衡



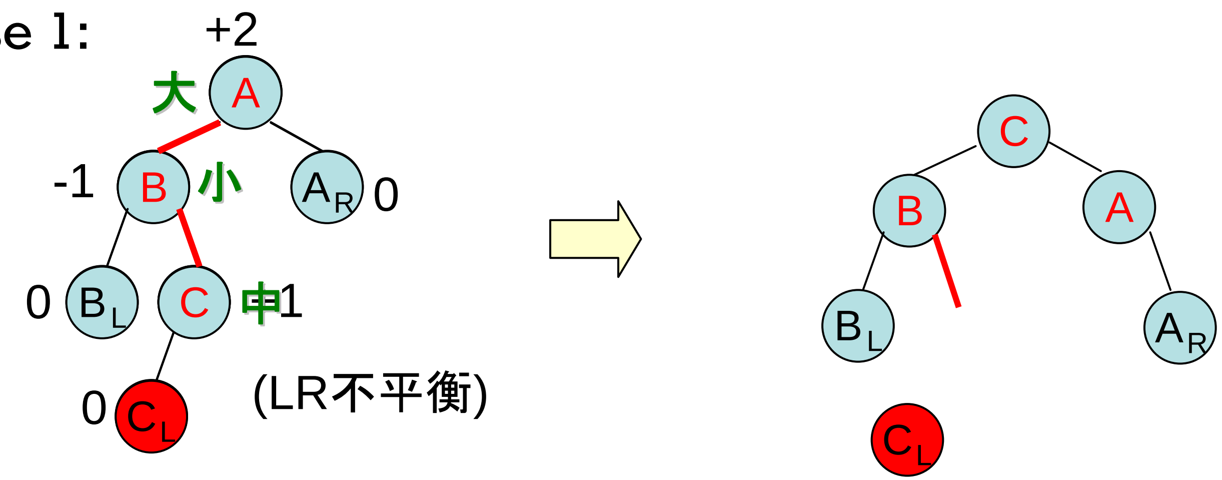
  - RL不平衡



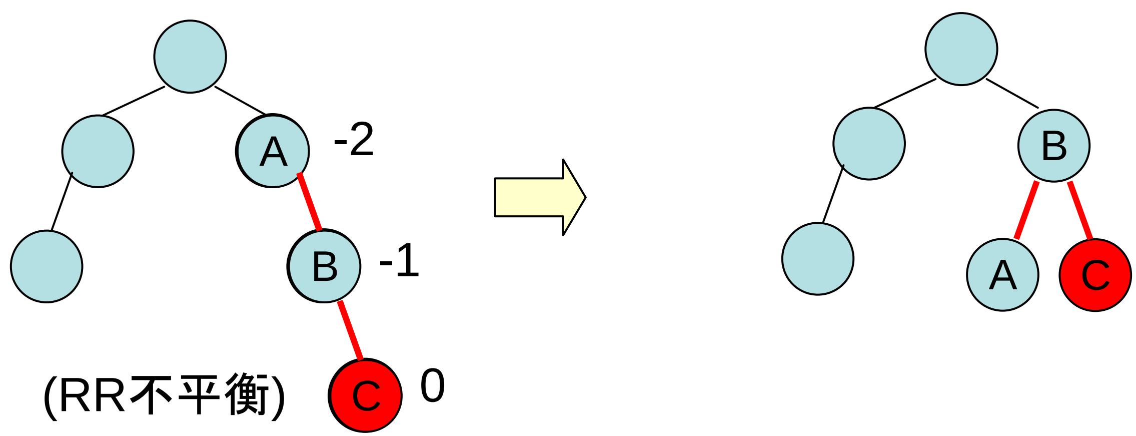
  - RR不平衡



- 範例 2：若插入資料時發生以下二種情況，該如何調整：
  - Case 1:



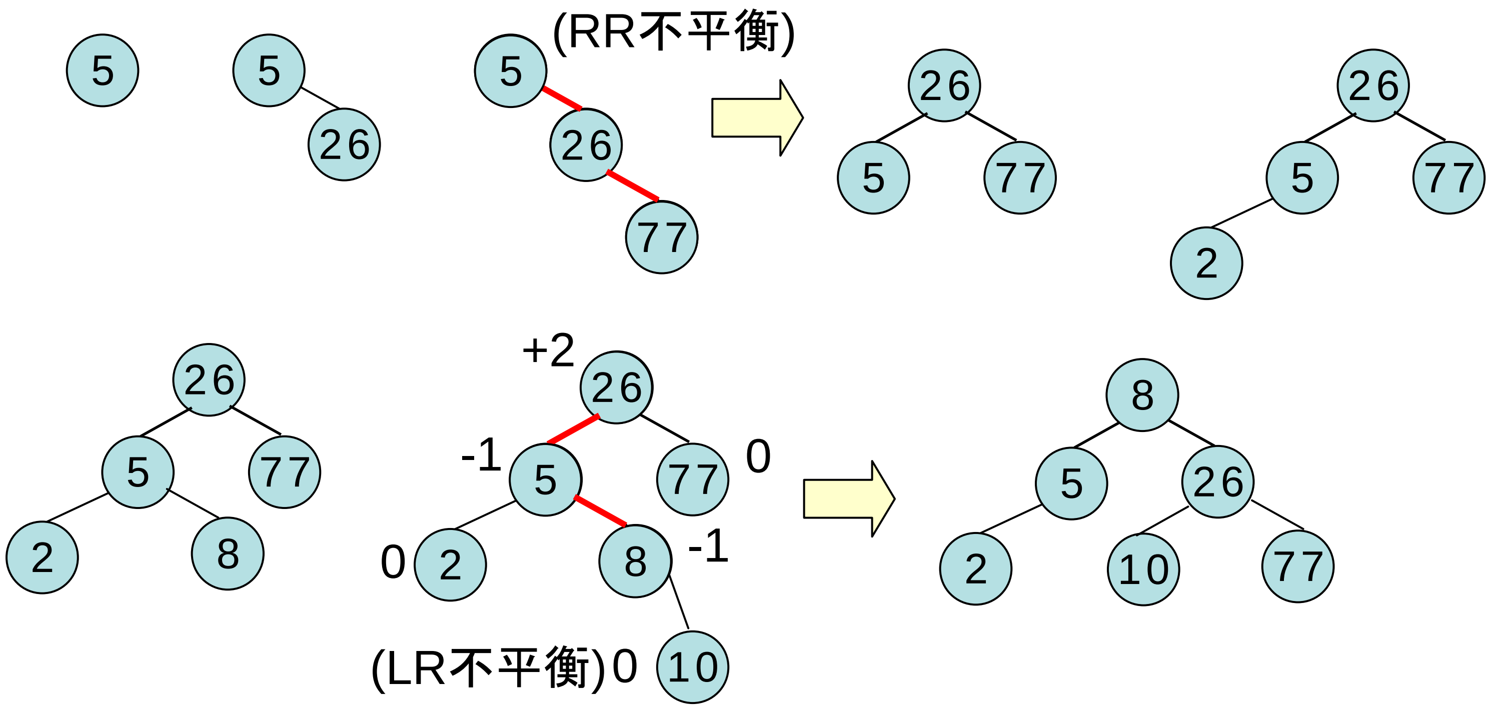
  - Case 2



- 範例 3：給予下列數字，請建立AVL Tree：

5, 26, 77, 2, 8, 10, 19, 12

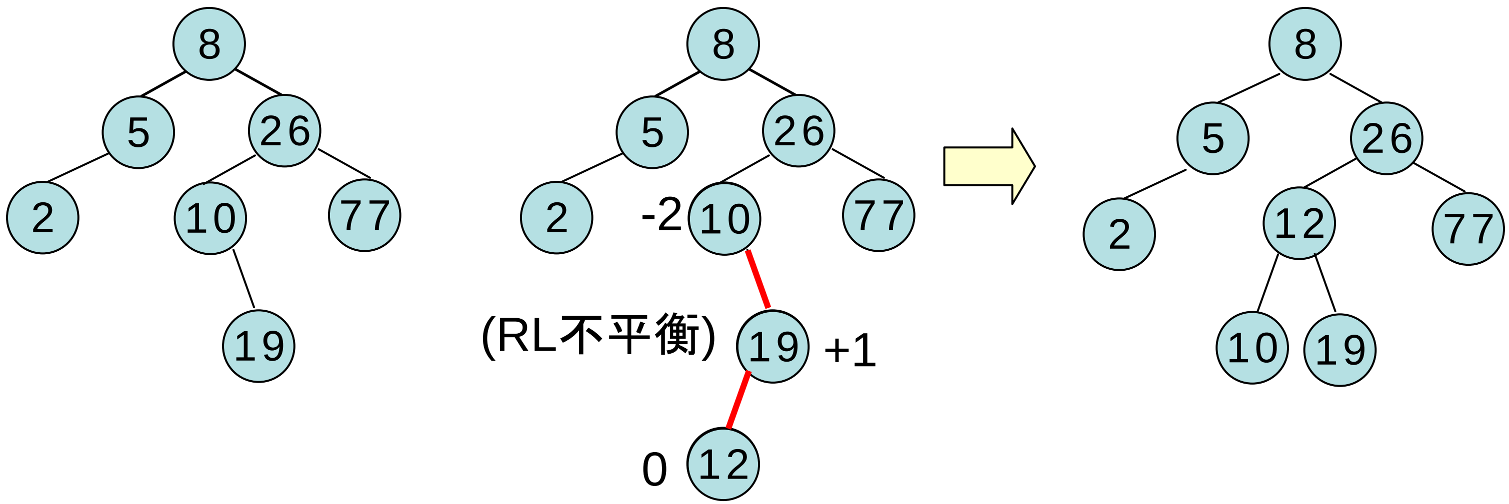
Ans:



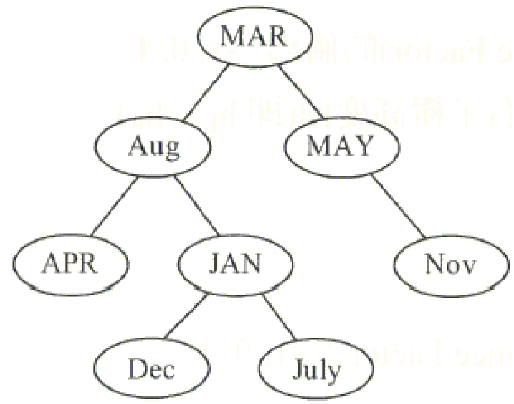
- 範例 3:給予下列數字, 請建立AVL Tree:

5, 26, 77, 2, 8, 10, 19, 12

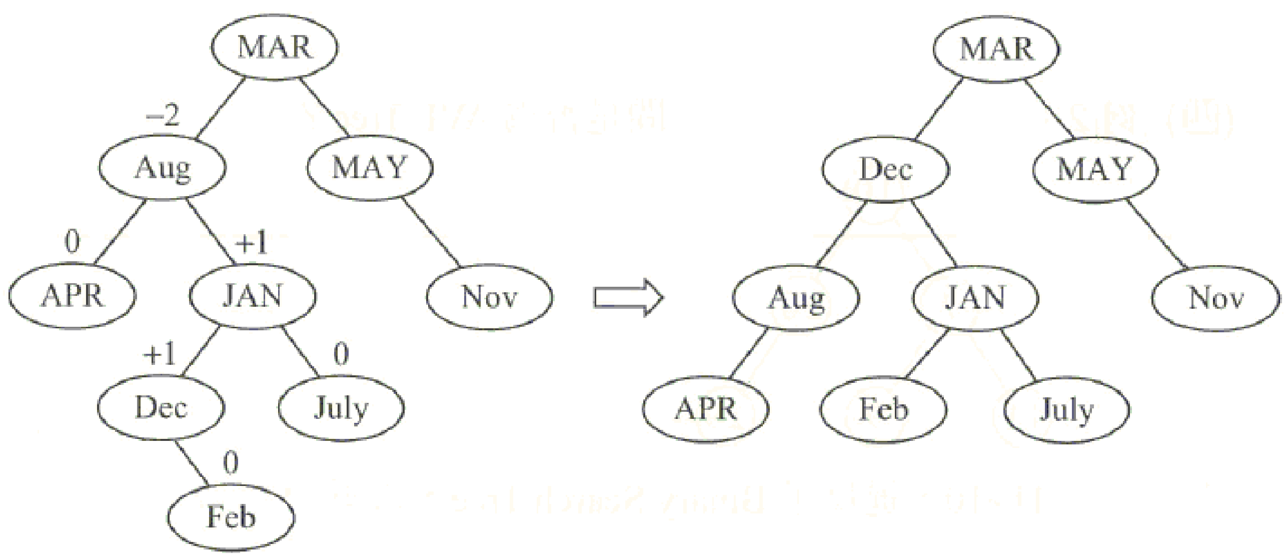
Ans: (續)



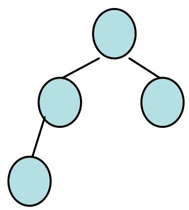
- 範例 4：下列英文月份的AVL Tree，若插入 “Feb” 之結果為何? (順序：A<B<C<…<Z)



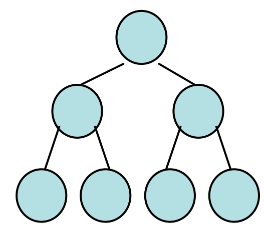
Ans:



- 定理：形成高度h的AVL Tree，所須要的最少節點個數為：$F_{h+2}-1$ (F：費式數)
- 範例 1：形成高度為3的AVL Tree，
  - 所需的最少節點個數為 4
  - 所需的最多節點個數為 7



$F_{h+2}-1 = F_{3+2}-1 = 4$



$2^3-1=7$

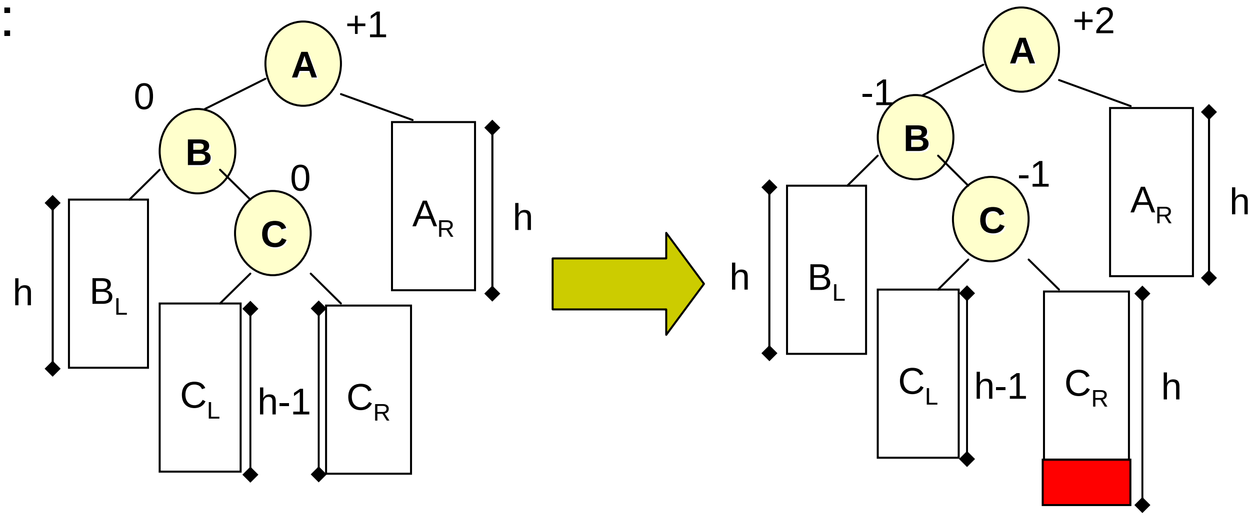
- 範例 2：一個AVL Tree有15個節點，則：
  - 最大高度為 5
    (用最少的點撐出最大的高度)

		h=5	h=6
n	…	7	8
F_n	…	13	21

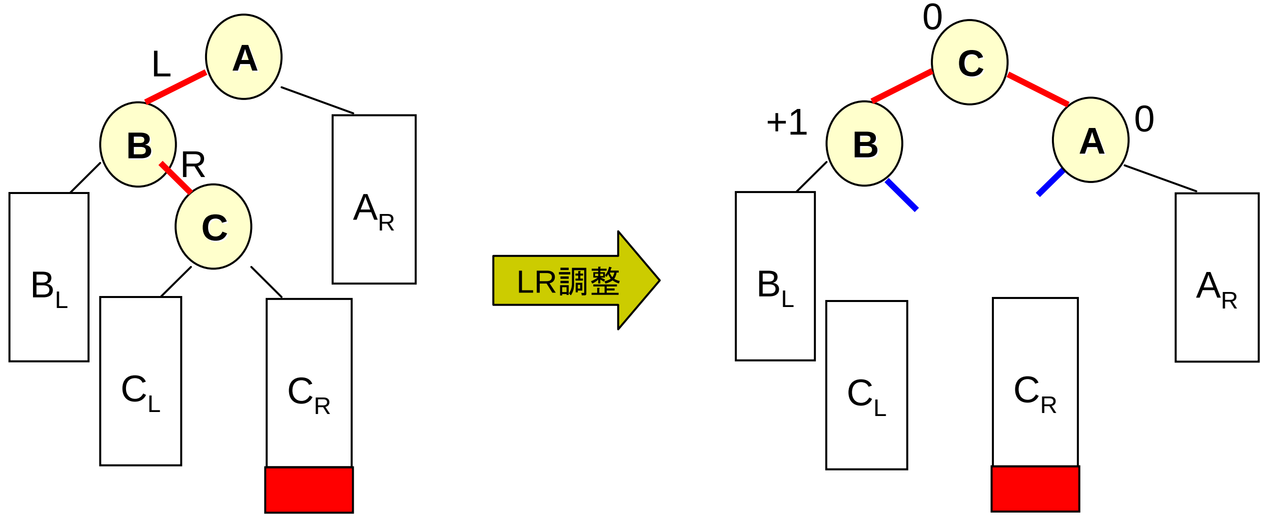
  - 最小高度為 4
    (完整二元樹)

$$\lceil \log_2(n+1) \rceil = \lceil \log_2(15+1) \rceil = 4$$

- 如:



如何調整成AVL Tree?



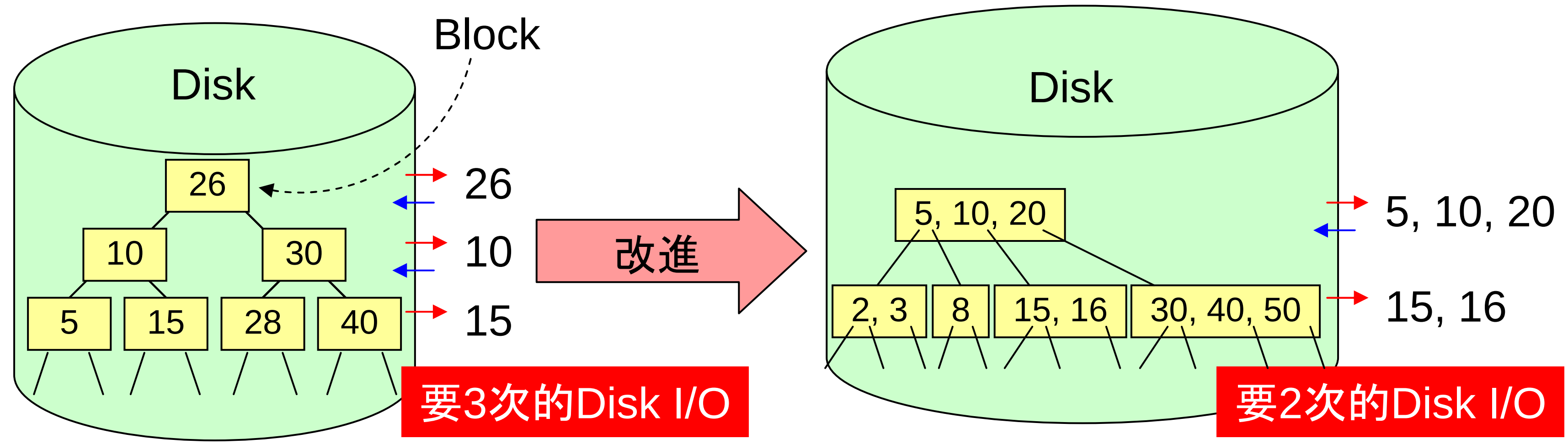
m-way Search Tree

- 主要用於**外部搜尋 (External Search)**
  - 先前的樹狀搜尋方法皆為內部搜尋 (Internal Search)
  - Internal Search:
    - Def: 資料量少, 可以一次全部置於**Memory**中進行search之工作
  - External Search:
    - Def: 資料量大, 無法一次全置於Memory中, 須藉助輔助儲存體 (E.g. Disk), 進行分段search之工作
- Tree Degree = m, 且m>2
  - 為何不用二元樹的結構?
    - 因為外部搜尋的資料量頗大, 若還是以二元樹的結構存放資料, 則樹的高度將很高, 資料搜尋將頗為費時。

- 目的: 提升External Search 的效益
  - 要提升External Search的效益，則需降低Disk I/O的次數
  - 要有效降低Disk I/O的次數，則需要降低Search Tree的高度
  - 要能夠降低Search Tree的高度，則需增大Tree Degree: m
- 範例: 找 “15” 這筆資料

: Disk out

: Disk in

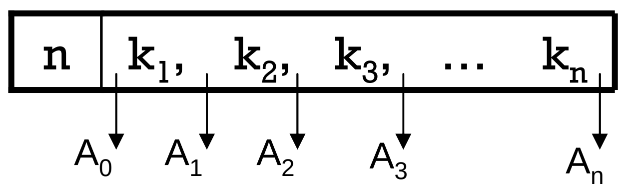


(B.S.T. 的 Degree最大為2時)

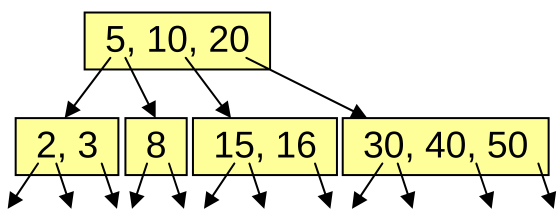
(Degree最大為4時)

- 若m為無窮大時，雖然高度只有一層，但因為Data量太大，無法一次放到Memory中。∴m以Memory Size為限。
  - 若所有資料可以一次全放到記憶體中，則為Internal Search的議題，不需用到外部搜尋的技術。

- Def: m-way Search Tree T是一個所有節點的分支度≤m的樹。T可以是空樹。若T不為空樹時，則具有下列性質：
  - 節點結構為：$n, A_0, (k_1, A_1), (k_2, A_2), \ldots, (k_n, A_n)$
    - 鍵值個數 $n \le m-1$
    - k_i: 鍵值 (資料)
    - A_i: 指標，指向<u>存放大小介於 k_i 與 k_{i+1} 之間的資料</u>之節點所在

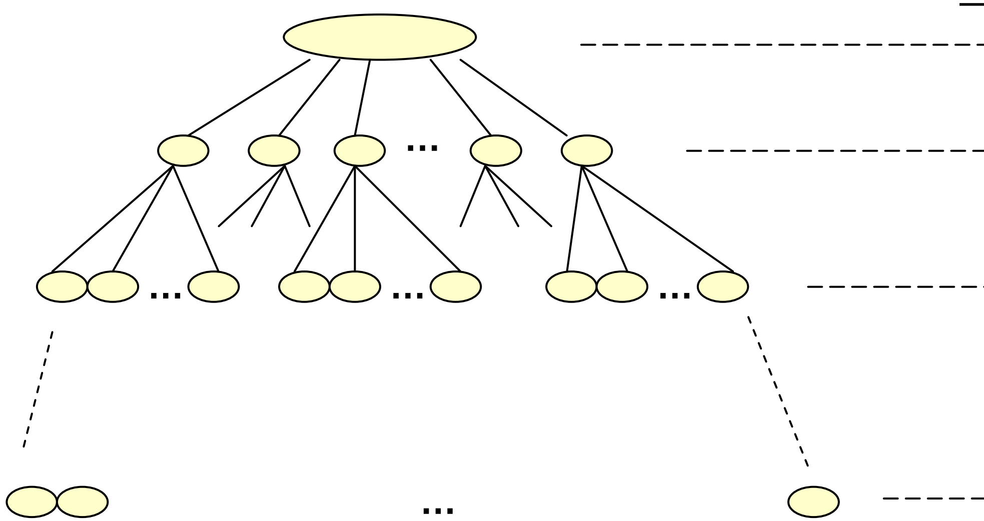


Ex:



  - 節點中的鍵值 (資料) 是由小至大排序的
  - 子樹同樣是m-way Search Tree (遞迴定義)
  - 子樹A_i內的資料均小於鍵值k_{i+1}；子樹A_n內的資料均大於鍵值k_n。

- 定理：高度為h的m-way Search Tree：
  - 最多的Node個數 = ?
  - 最多的Key (Data) 個數 = ?



Level	每層最多的Node個數
1	
2	
3	
⋮	⋮
h	

∴高度為h的m-way Search Tree最多的Node個數 $= m^0 + m^1 + m^2 + \ldots + m^{h-1}$
$= (m^h - 1)/(m-1)$

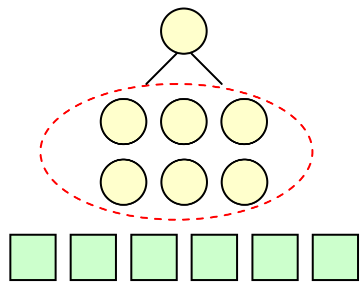
∴高度為h的m-way Search Tree最多的Data個數 $= (m-1) \times (m^h - 1)/(m-1)$
$= (m^h - 1)$

- m-way Search Tree雖利用增大Tree的Degree來降低Search Tree的高度，以減少Disk I/O 的次數，但若此Tree不平衡，也可能會變相地增加Disk I/O的次數。
- 因此發展出B-tree of order m。

B-Tree

- Def: 是一個Balanced m-way search tree。可以為空，若不為空，則滿足：
  - Root至少有2個Childs
  - 除了Root及Failure Node之外，其餘Node的分支度介於$\lceil m/2 \rceil$及m之間
  - 所有的Failure Node皆位於同一Level。(即：所有葉子節點位於同一Level)

- 範例：有一B-tree of order m如下：



- 若m = 3 (即：B-tree of order 3)
  - **Root**：$2 \le$ degree ≤ 3；
  - **非Root與非Leaf的節點**：$\lceil m/2 \rceil \le$degree$\le$m $\Rightarrow 2 \le$ degree ≤ 3
  - B-tree of order 3又稱 2-3 tree
- 若m = 4 (即：B-tree of order 4)
  - **Root**：$2 \le$ degree ≤ 4；
  - **非Root與非Leaf的節點**：$\lceil m/2 \rceil \le$degree$\le$m $\Rightarrow 2 \le$ degree ≤ 4
  - B-tree of order 4又稱 2-3-4 tree
- 若m = 5 (即：B-tree of order 5)
  - **Root**：$2 \le$ degree ≤ 5；
  - **非Root與非Leaf的節點**：$\lceil m/2 \rceil \le$degree$\le$m $\Rightarrow 3 \le$ degree ≤ 5

B-Tree of order m之相關定理

- 定理：高度為h的B-Tree：
  - 最多的Node個數 = ?
  - 最多的Key (Data) 個數 = ?

由於B-Tree中，每一個節點的Degree最大皆可到m (m可自定)
所以其結果等同於m-way Search Tree的分析結果!!!

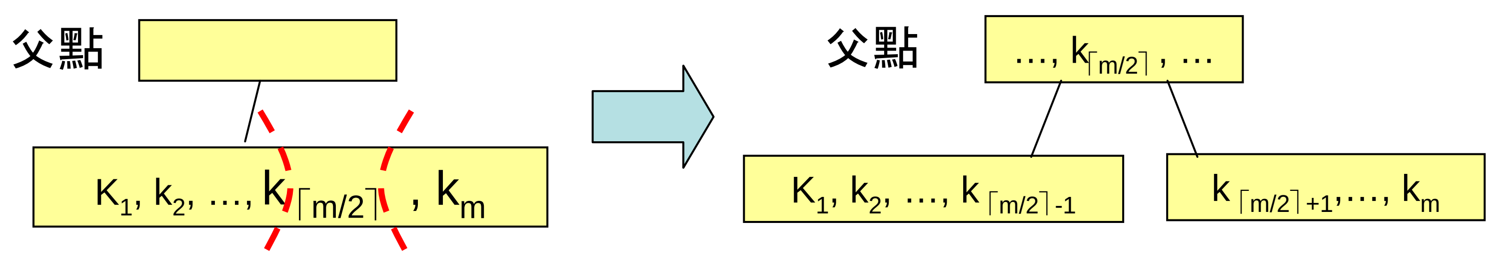
B-Tree之Insert動作 (假設要插入資料x)

- Step:
  ① 先做Search，以找到適當的插入節點，將x放入該節點中

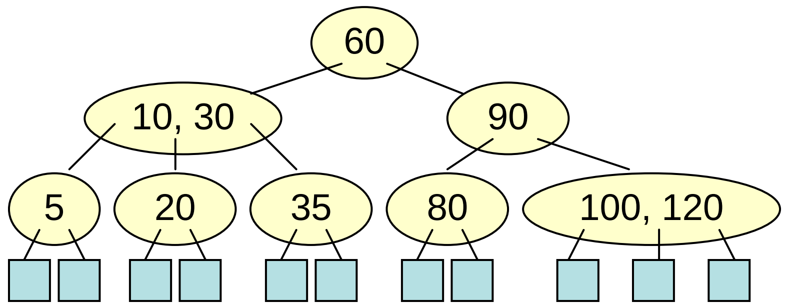
  ② 檢查該節點有無Overflow

  - 無Overflow (key數≤m-1)，則OK並跳出。
  - 有Overflow (key數≥ m)則：
    - 作 “Split” 處理
    - 針對父點，goto ②　　//即：作完Split後，針對父點檢查是否有Overflow

- 如何做 “Split” 處理
  - 將第⌈m/2⌉個鍵值上拉到父點，其餘鍵值分成左、右兩個子點。



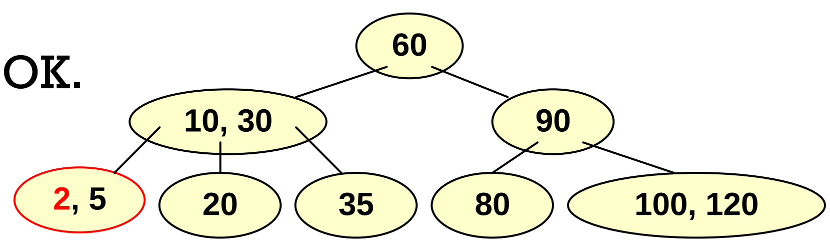
- 範例：有一B-tree of order 3如下，若連續Insert “2”, “130”, “8”會得到什麼結果。



Ans:

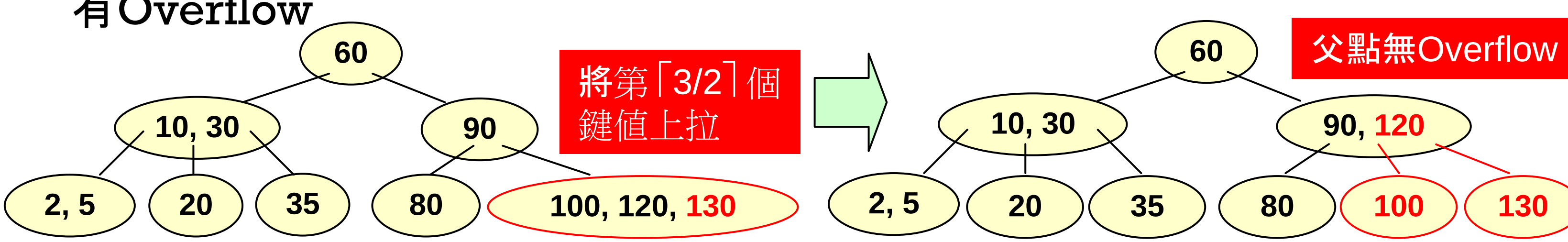
- Insert “2”

  無Overflow, ∴ OK.



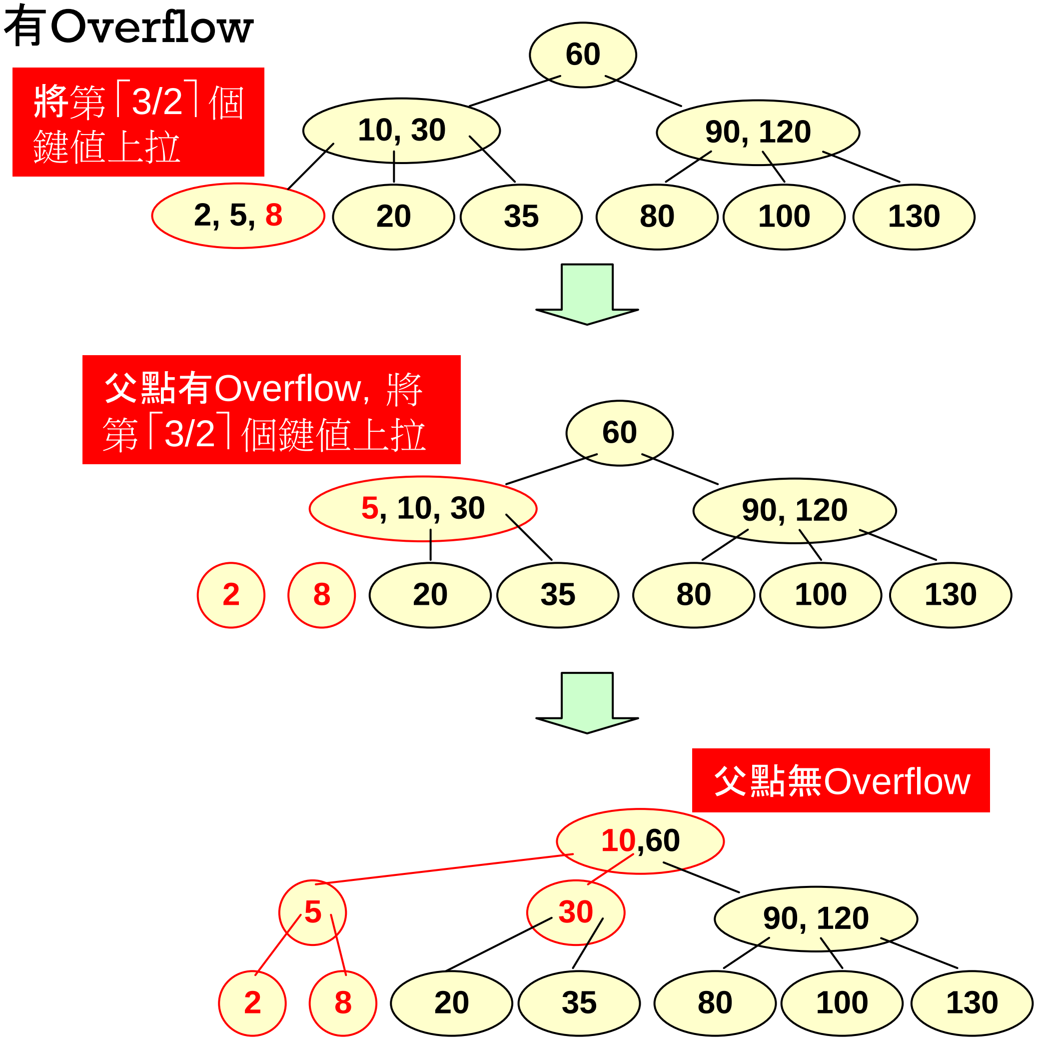
- Insert “130”

  有Overflow



- Insert “8”

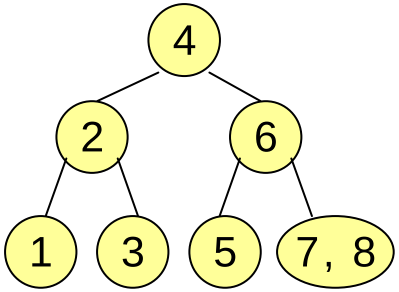
  有Overflow



※練習範例※

- 給予1, 2, 3, 4, 5, 6, 7, 8, 請建立2-3 tree (B-tree of order 3)

Ans:



B-Tree之Delete動作 (假設要刪除資料x)

- Step:
  - 先做Search，以找到x所在之節點
  - 分成下列兩個Case來處理：

    Case 1: x位於Leaf Node

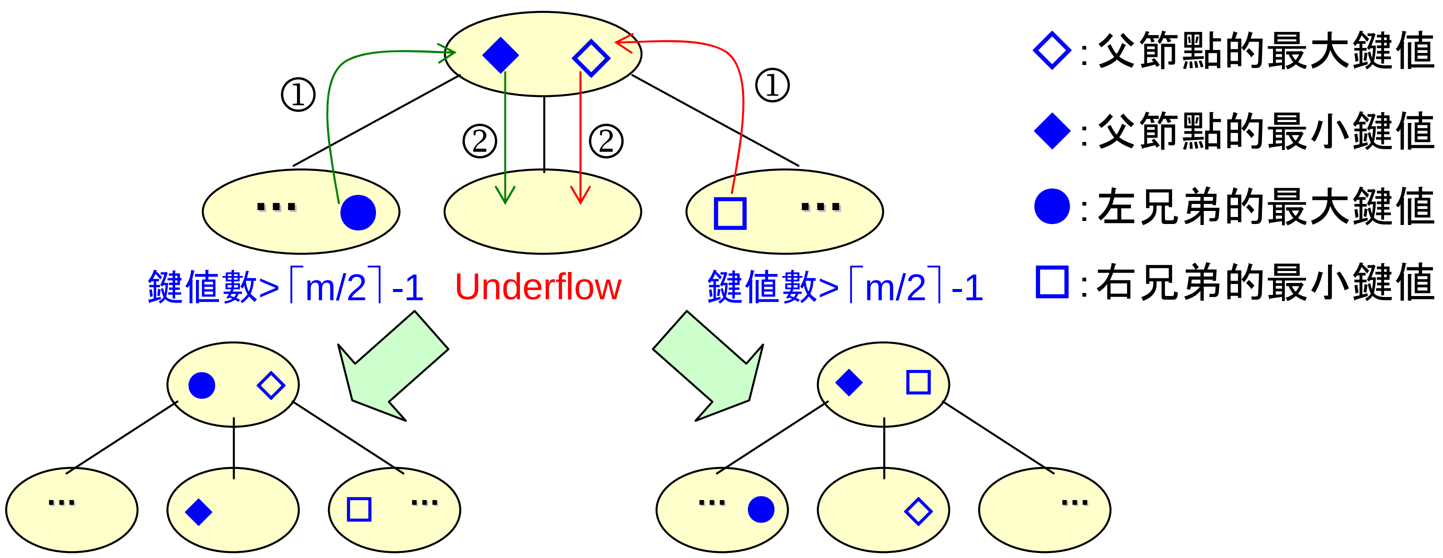
    Case 2: x位於Non-leaf Node

Case 1: x位於Leaf Node

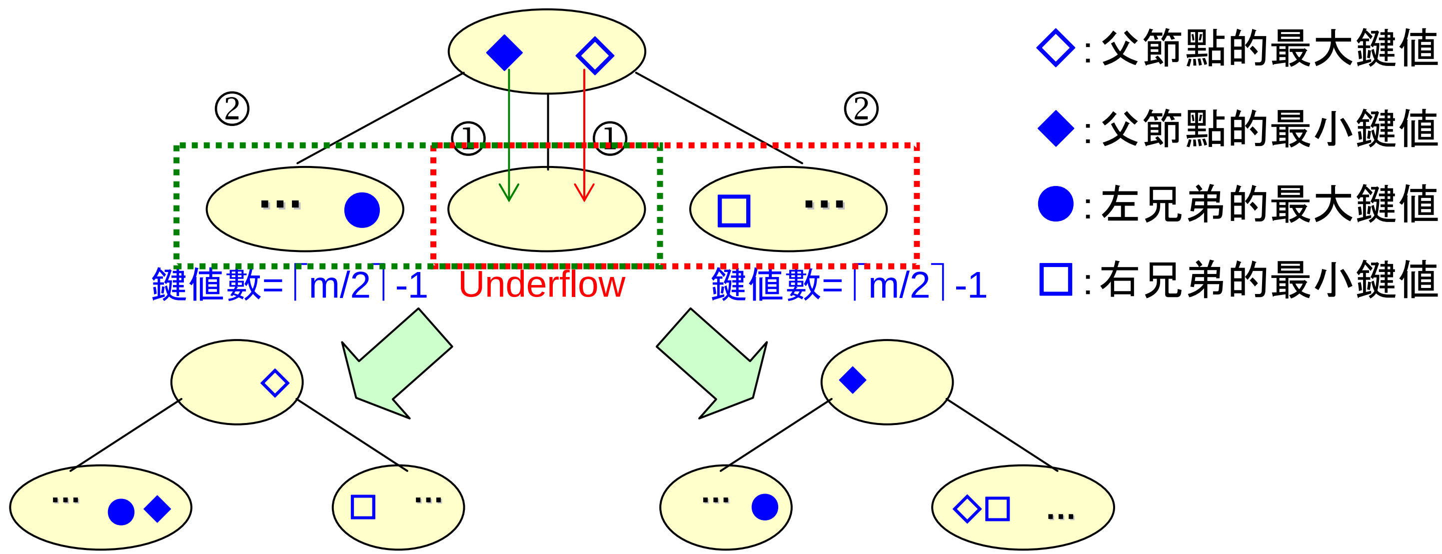
- 需進行下列步驟:
  - ① Delete x
  - ② 檢查該Node有無Underflow
    - 沒有Underflow (即:鍵值數目 ≥⌈m/2⌉-1) 則OK，結束工作。
    - 有Underflow (即:鍵值數目 <⌈m/2⌉-1)，則:
      - 先嘗試作 "**Rotation**"，若完成工作則結束。
      - 若無法做"Rotation"，則作 "**Combination**"。Combination做完，針對父點，goto ②。

※ B-tree of order m中，除了樹根與失敗節點之外，其它的Internal Node的degree是介於⌈m/2⌉~m之間，∴每個Internal Node內的鍵值<u>最多有m-1個鍵值，最少有⌈m/2⌉-1個鍵值</u>!!

- Rotation的處理，可分成：
  - 右旋：
    ① 將右兄弟的最小鍵值傳至父點，
    ② 將原父點的最大鍵值傳給Overflow的節點做為其最大鍵值
  - 左旋：
    ① 將左兄弟的最大鍵值傳至父點，
    ② 將原父點的最小鍵值傳給Overflow的節點做為其最小鍵值
  - 這些Rotation的結果可保有Binary Search Tree的鍵值順序，即：左子樹(小) - 樹根(中) - 右子樹(大)

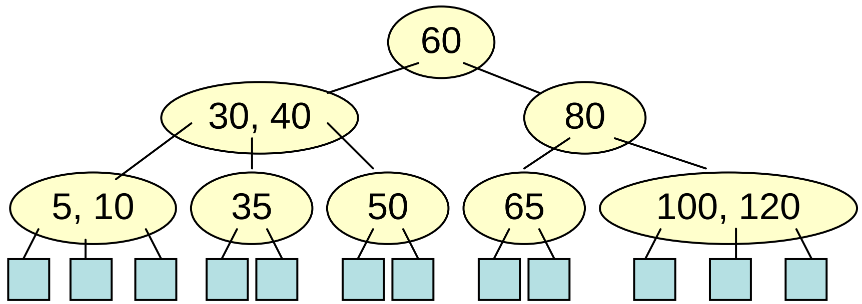


- Combination的處理，可分成：
  - 右合併：
    ① 將父點右邊的最大鍵值傳給Overflow的節點做為其最大鍵值
    ② 將Overflow的節點與其<u>有相同鍵值來源之兄弟</u>合併成一個節點
  - 左合併：
    ① 將父點左邊的最小鍵值傳給Overflow的節點做為其最小鍵值，
    ② 將Overflow的節點與其<u>有相同鍵值來源之兄弟</u>合併成一個節點



- 這些Combination的結果可保有Binary Search Tree的鍵值順序，即：
  左子樹(小) - 樹根(中) - 右子樹(大)

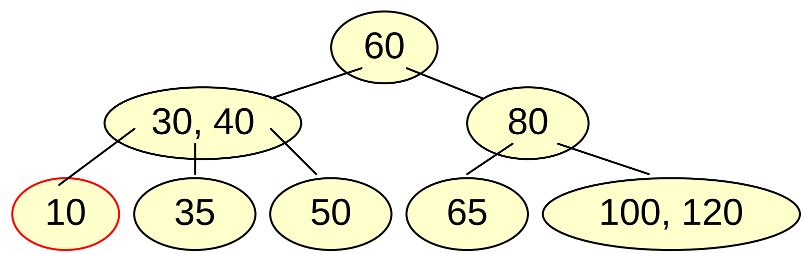
- 範例 1：有一B-tree of order 3如下，若連續Delete “5”，“65”，“50”會得到什麼結果。



Ans: (p.s.: $2 \le$ degree $\le 3 \Rightarrow 1 \le$ 鍵值數量 ≤ 2)

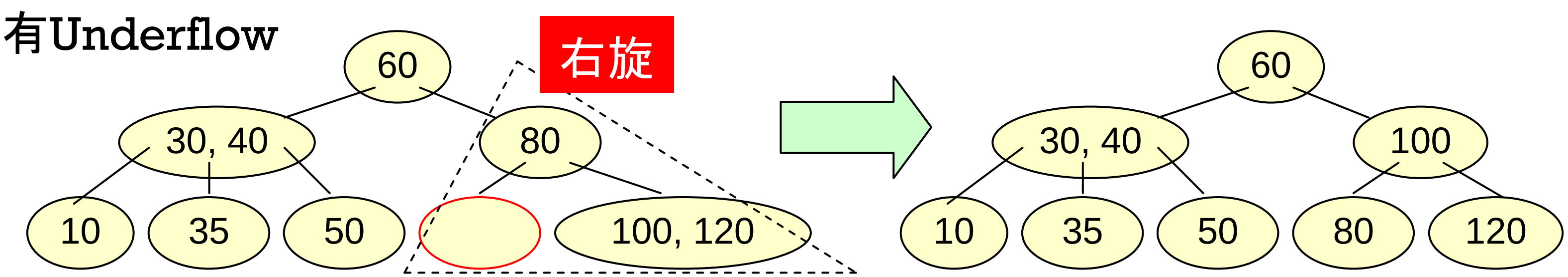
- Delete “5”

  無Underflow，∴OK.



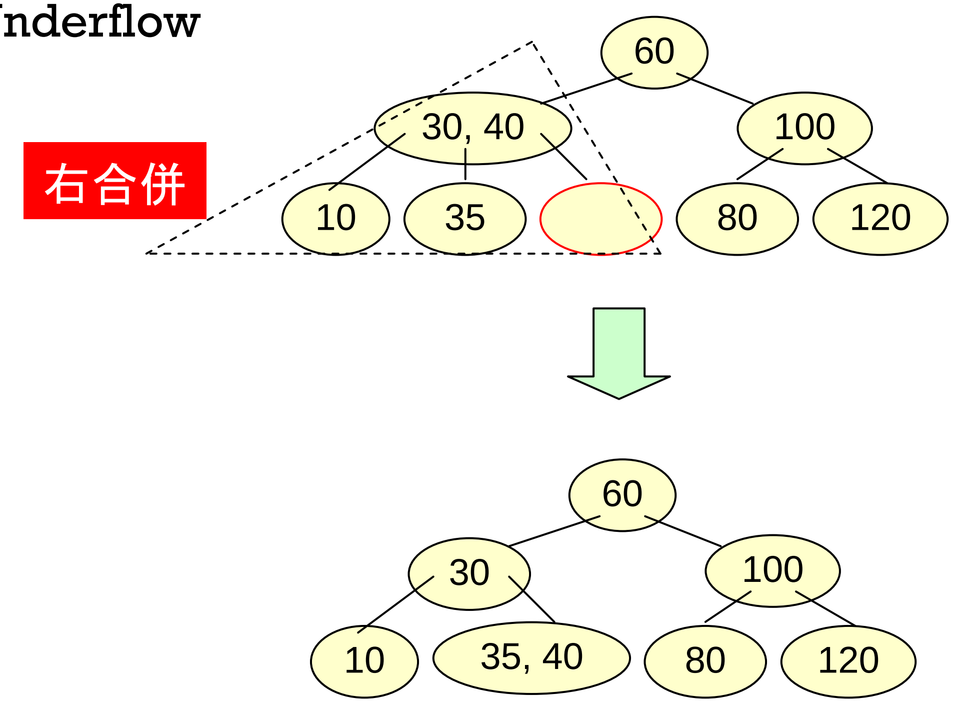
- Delete “65”

  有Underflow

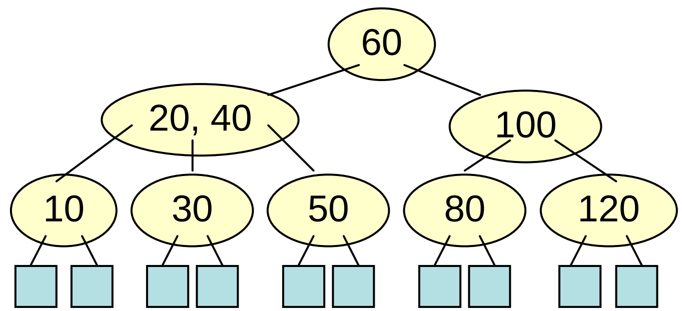


- Delete “50”

  有Underflow

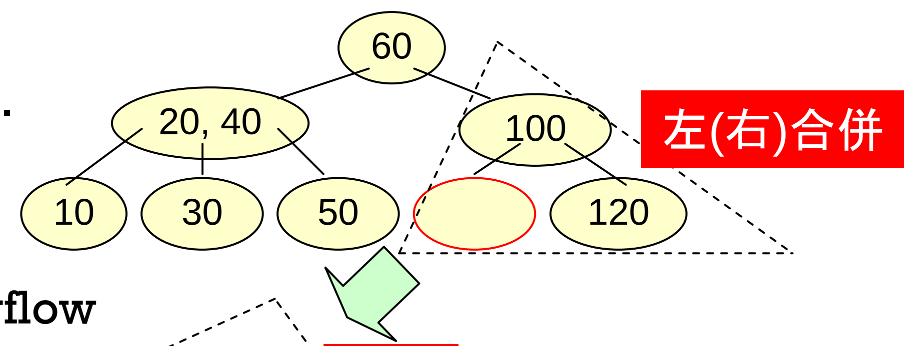


- 範例 2：有一B-tree of order 3如下，若連續Delete “80”，“10”會得到什麼結果。

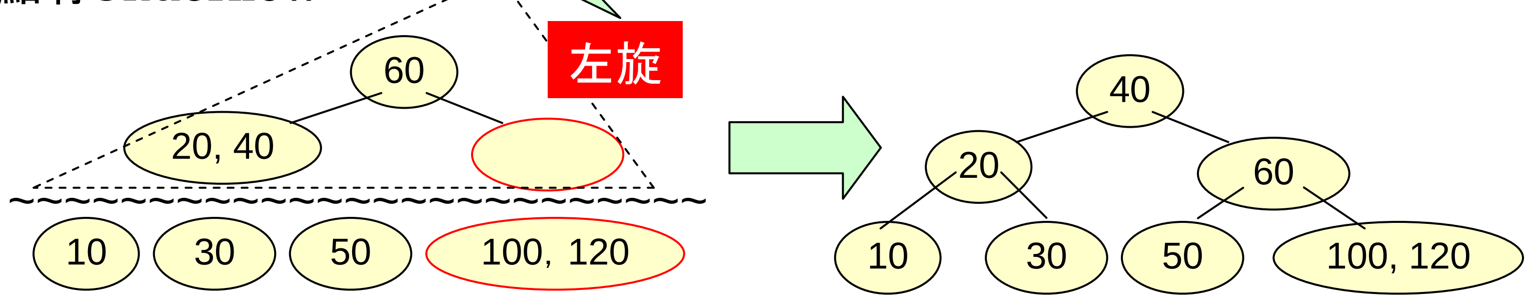


Ans: (p.s.: $2 \le$ degree $\le 3 \Rightarrow$ 1 ≤ 鍵值數量 ≤ 2)

- Delete “80”
  有Underflow.

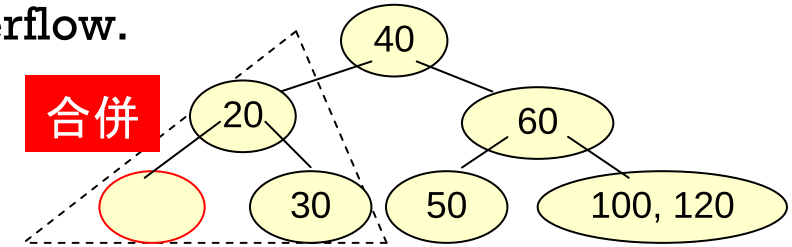


- 父點有Underflow

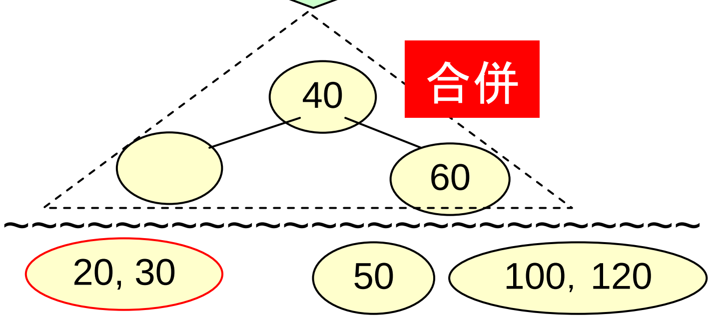


- Delete “10”

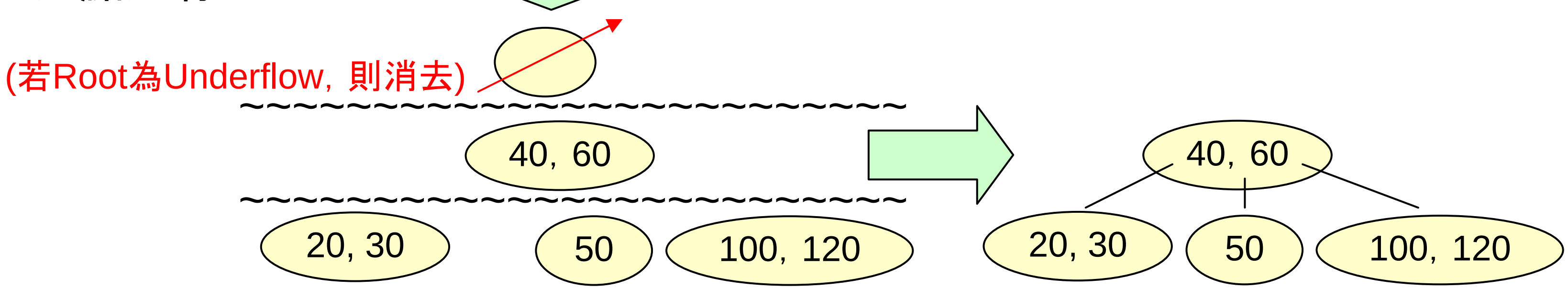
有Underflow.



- 父點有Underflow



- 父點又有Underflow



Case 2: x位於Non-leaf Node

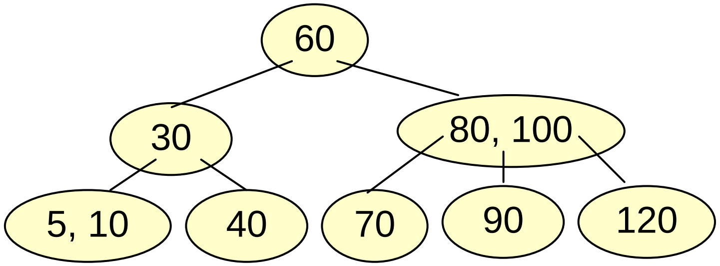
- 需進行下列步驟:

  ① (a). 以x的右子樹中之最小鍵值取代x; 或是

  (b). 以x的左子樹中之最大鍵值取代x

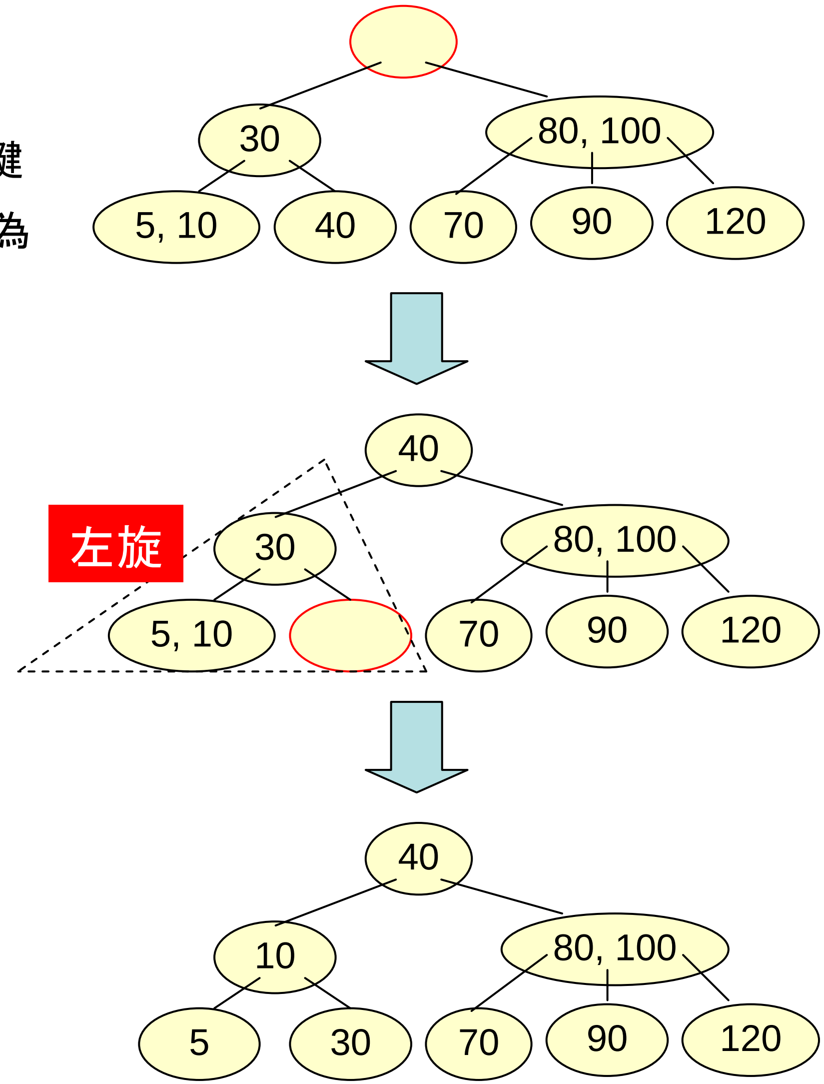
  ② 返回Case 1作後續處理。

- 範例 1 : 有一B-tree of order 3如下，若連續Delete “60”，“100”會得到什麼結果。

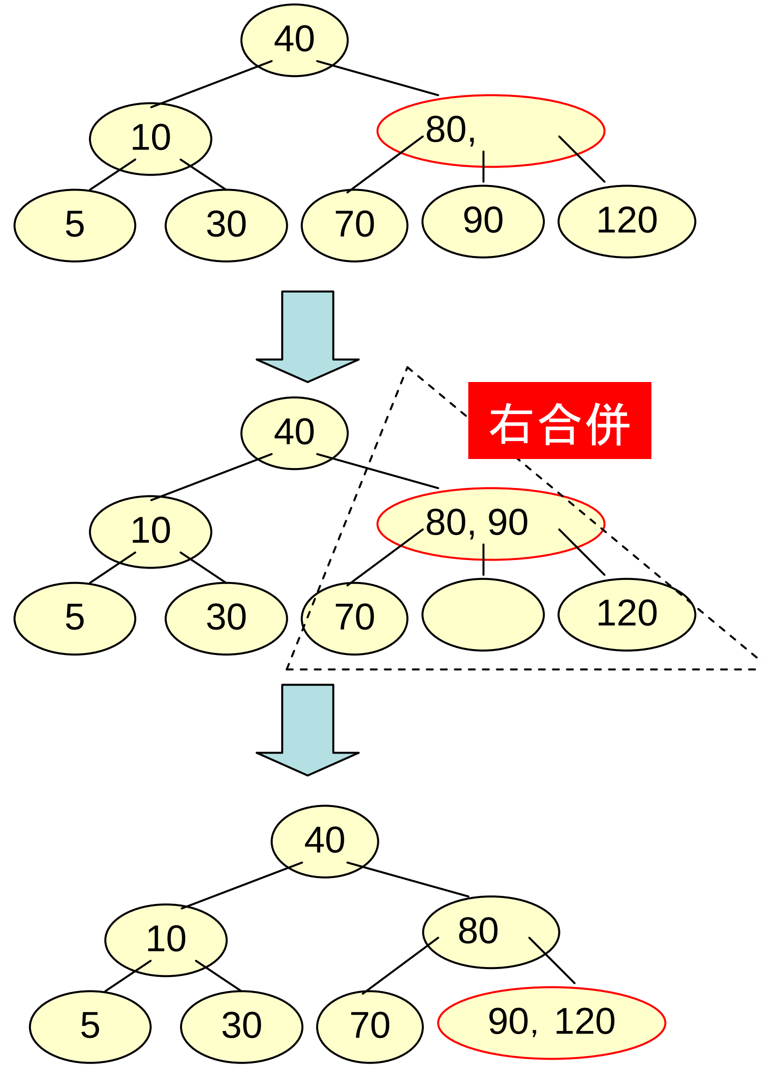


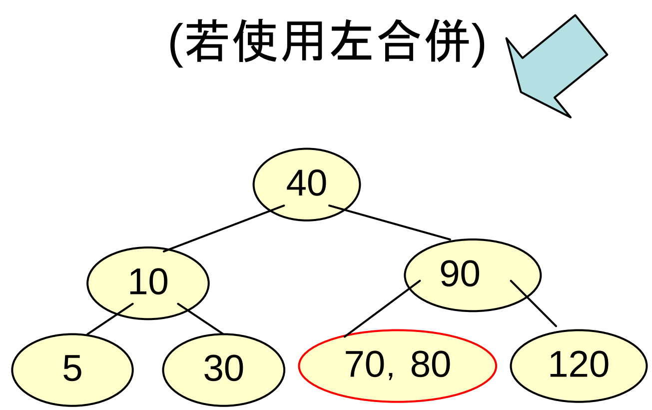
Ans: (p.s.: $2 \le$ degree $\le 3 \Rightarrow 1 \le$ 鍵值數量 ≤ 2)

- Delete “60”
  - 要選擇右子樹最小 or 左子樹最大的鍵值來取代被刪除的資料，以最方便的為優先!!
  - Rotation > Combination
- 返回Case 1做後續處理



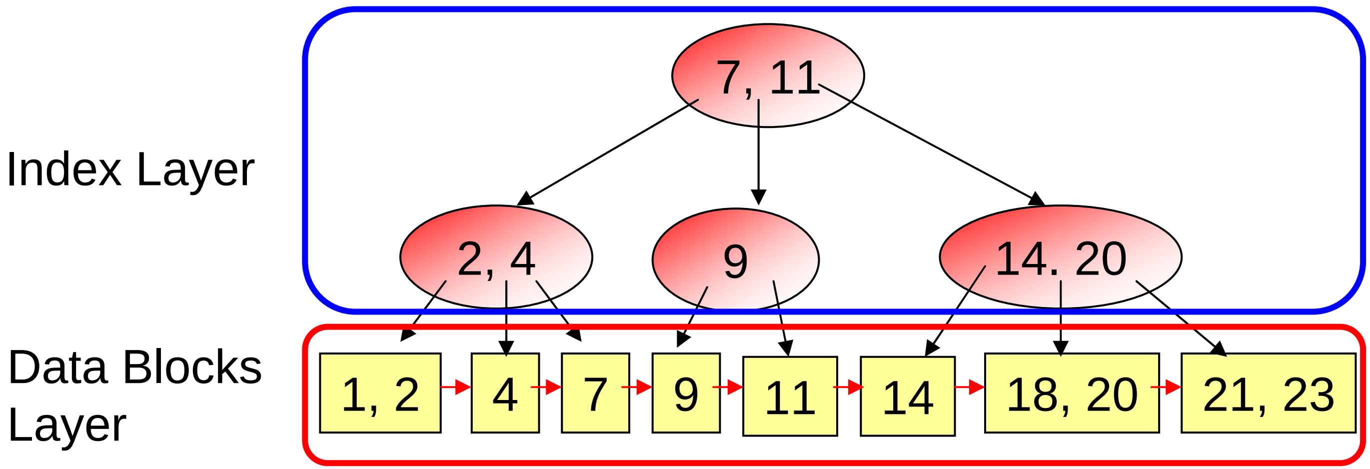
- Delete “100”
  - 要選擇右子樹最小 or 左子樹最大的鍵值來取代被刪除的資料，以最方便的為優先!!
  - Rotation > Combination
- 返回Case 1做後續處理





■ B+ Tree

- Def:
  - 是B Tree之變形，也是用於External Search/Sort上。
  - 可以支援ISAM (Index Sequential Access Method)之實施，常用於DBMS內層製作。
- 以Order-3為例：



- B^+ tree分為兩個Layer
  - **Index Layer:**
    - 採B tree結構。僅用來存放索引，以幫助User正確地找到所需之Data Block。
  - **Data Blocks Layer:**
    - 用以存放Data，且Blocks之間以Link List串連。
    - Data Block存放資料時，通常不會受到B tree的Order限制。因此，每個Data Block最多要塞多少筆Data可自訂。當然也可以和Index Layer內的節點一致。
- 何謂**Index Sequential Access**?
  - 透過Index找到對應的起始Data Block之後，即可循序讀取其它資料。
  - Ex: 找出大於等於7~小於20的資料。